Full LR(1) Parser Generator Hyacc

Implementation

Xin Chen, Ph.D.

University of Hawai'i

Tutorial at the International Symposium on Code Generation and Optimization (CGO 2012), San Jose, CA

March 31, 2012

Outline

- Overview
- Architecture
- Parse engine
- Storing the parsing table
- Handling precedence and associativity
- Error handling
- Data structures
- Future work

I. Overview

- Hyacc
- Implemented in ANSI C
- Developed between 8/29/2006 and early 2009.

8/29/2006

Task: Write a Yacc that combines compatible states and removes unit products.

Yacc: Given a grammar, generate a parsing machine, then generate a compiler that embeds the parsing machine.

Master Plan - Steps of implementation:

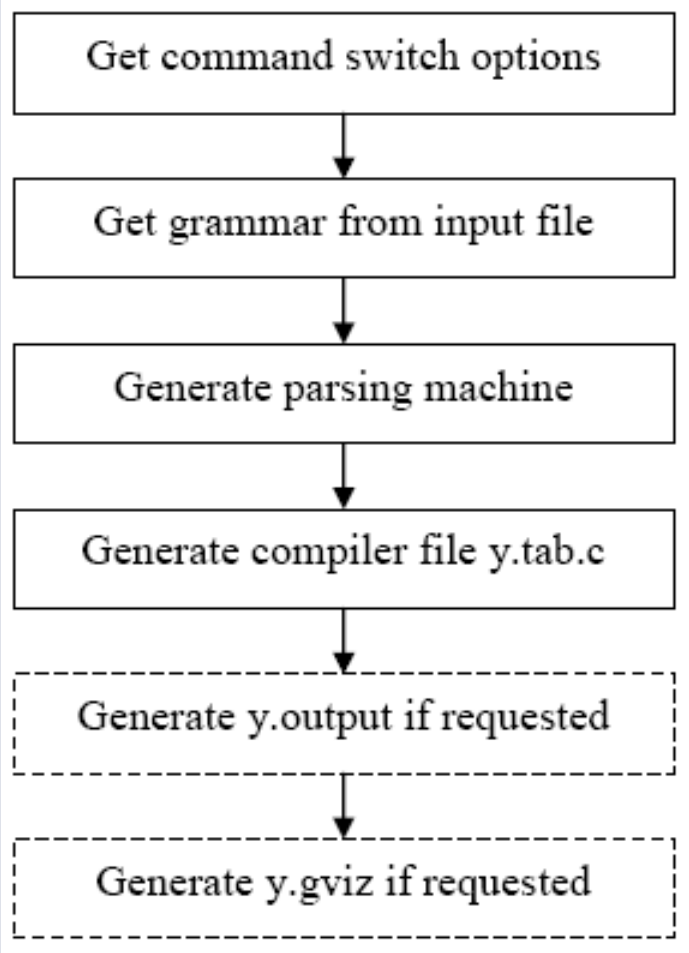
- start with a fixed simple grammar (E->E+T|T, T->T*a|a).
1. Create LR(1) parsing machine.
2. Add in 1) the function of combining compatible states.
3. Add in 2) the post-modification of removing unit productions.
- Try the above with different fixed grammars to validate the program.
4. Use Yacc input file to provide the grammar. This needs code to parse the Yacc input file.
5. Add the code of generating a compiler by embedding the parsing machine.

NOTE A: Tasks 1,2,3,4,5 are actually independent tasks.
NOTE B: So I'll try to finish 1,2,3 first, and then 4 and 5 if I have more time.

Here is step 1 preliminary design. Show Detail

II. Architecture

- Task flow

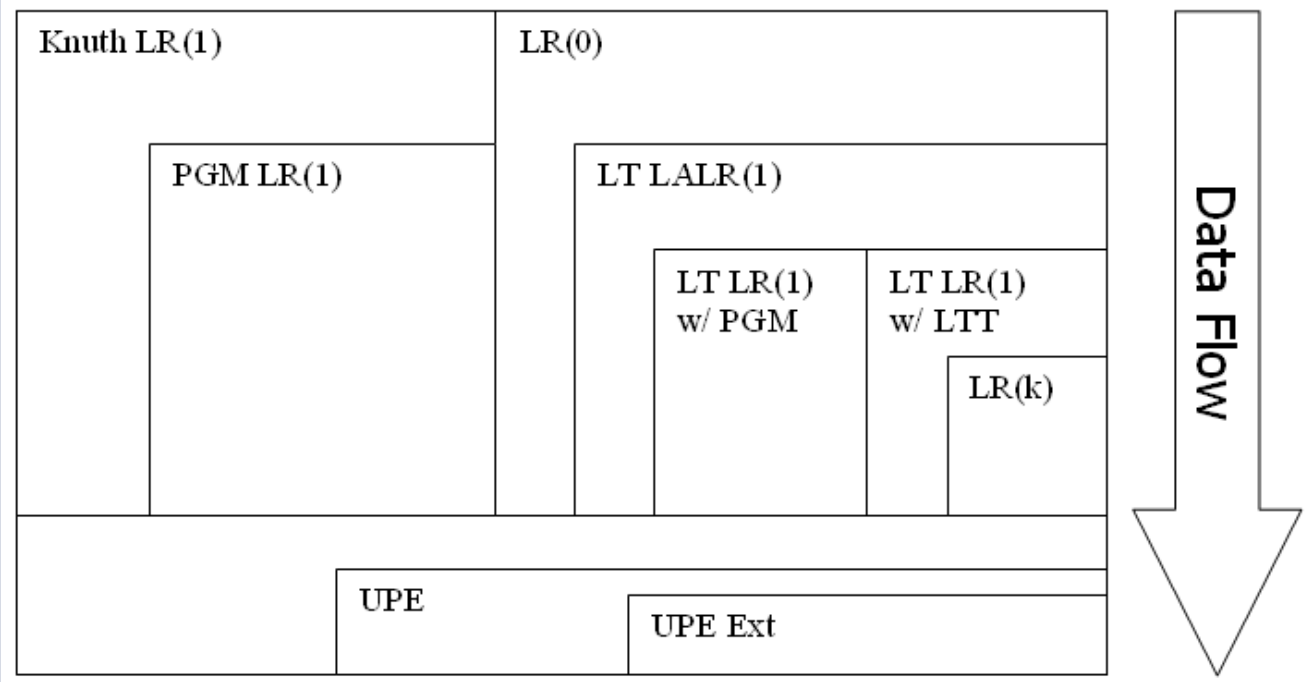


II. Architecture

- Algorithms
  - Knuth LR(1): Knuth canonical LR(1) algorithm (Knuth)
  - PGM LR(1): Practical General Method (Pager)
  - UPE: Unit Production Elimination (Pager)
  - UPE Ext: UPE Extension (Chen)
  - LR(0)
  - LT LALR(1): LALR(1) based on Lane-tracing (L.T.) (Pager)
  - LT LR(1) w/ PGM: LR(1) based on L.T. using PGM (Pager)
  - LT LR(1) w/ LTT: LR(1) based on L.T. using Lane table (Pager)
  - LR(k): Edge-Pushing Algorithm (Chen)

II. Architecture

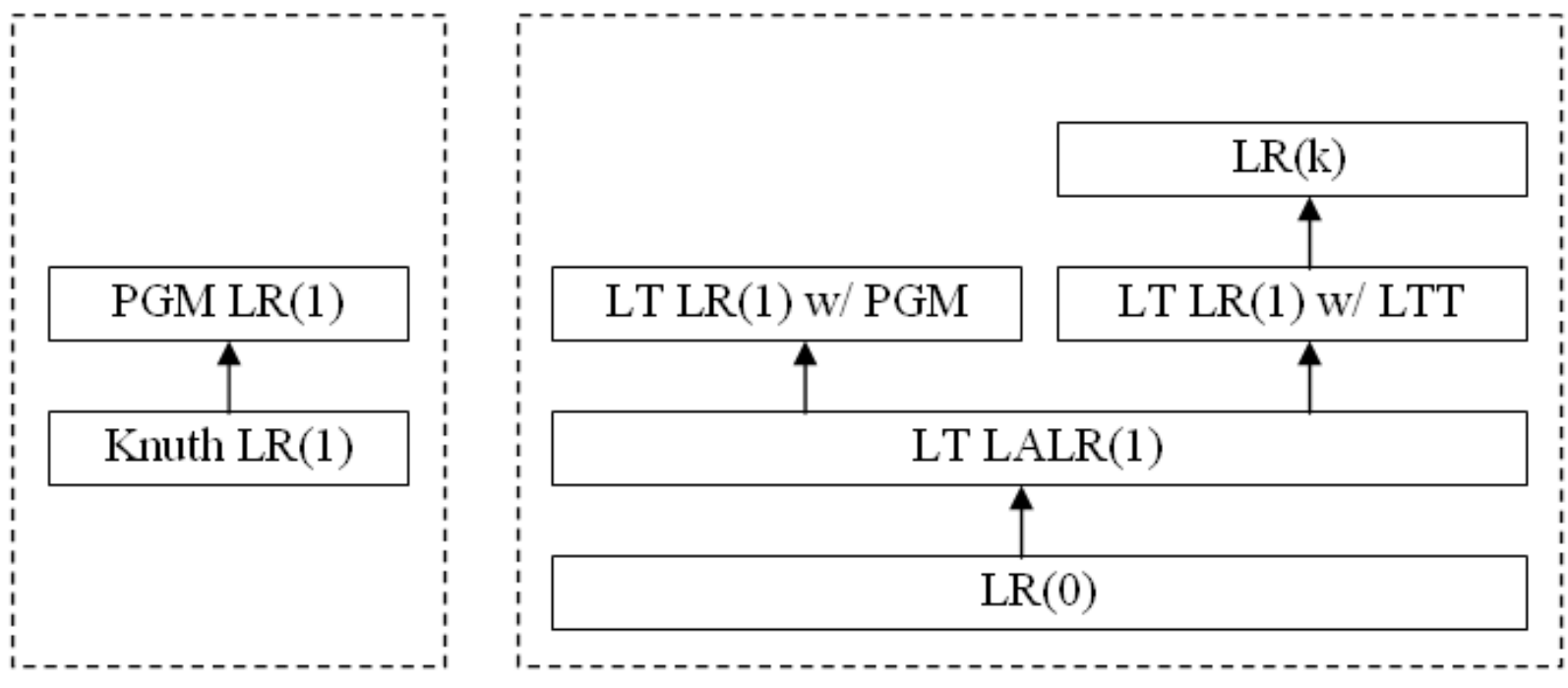
- Algorithm stack: view of data/grammar processing



PGM: Practical General Method; LT: Lane-Tracing; LTT: Lane-Tracing Table;
UPE: Unit Production Elimination; UPE Ext: UPE Extension

II. Architecture

- Algorithm stack: view of implementation



PGM: Practical General Method; LT: Lane-Tracing; LTT: Lane-Tracing Table;
UPE: Unit Production Elimination; UPE Ext: UPE Extension

III. Parse Engine

- LR(1) Parse Engine
  - Similar to Yacc parse engine
- LR(k) Parse Engine
  - Added an entry for reduce/reduce conflict in LR(1) parse engine

Table 1. The hyaccpar LR(1) parse engine algorithm

Algorithm 1: The hyaccpar LR(1) parse engine algorithm.

```
1  Initialization:
2  push state 0 onto state_stack;

3  while next token is not EOF do {
4    S ← current state;
5    L ← next token/lookahead;
6    A ← action of(S, L) in parsing table;
7    if A is shift then {
8      push target state on state_stack,
9        pop lookahead symbol;
10     update S and L;
11   } else if A is reduce then {
12     output code for this reduction;
13     r1 ← LHS symbol of reduction A;
14     r2 ← RHS symbol count of A;
15     pop r2 states from state_stack,
16       update current state S;
17     Atmp ← action for (S, r1);
18     push target goto state Atmp to state_stack;
19   } else if A is accept then {
20     if next token is EOF then {
21       is valid accept, exit;
22     } else {
23       is error, error recovery or exit;
24     }
25   } else {
26     is error, do error recovery;
27   }
28 }
```

Table 2. The hyaccpark LR(k) parse engine algorithm

Algorithm 2: The hyaccpar LR(k) parse engine algorithm.

```
1  Initialization:
2  push state 0 onto state_stack;

3  while next token is not EOF do {
4    S ← current state;
5    L ← next token/lookahead;
6    A ← action of(S, L) in parsing table;
7    if A is shift then {
8      push target state on state_stack,
9        pop lookahead symbol;
10     update S and L;
11   } else if A is reduce then {
12     output code for this reduction;
13     r1 ← LHS symbol of reduction A;
14     r2 ← RHS symbol count of A;
15     pop r2 states from state_stack,
16       update current state S;
17     A_tmp ← action for (S, r1);
18     push target goto state A_tmp to state_stack;
19   } else if A is reduce/reduce conflict then {
20     let k = 2;
21     while true do
22        L_next ← next lookahead;
23        if L_next == EOF then
24            Report error and exit;
25        else
26            In LR(k) parsing table, find entry A_next ← ((S, L), L_next);
27            if A_next is reduce/reduce conflict then
28                L ← L_next;
29                k ← k + 1;
30            else
31                do reduce;
32                break out of while loop;
33   } else if A is accept then {
34     if next token is EOF then {
35       is valid accept, exit;
36     } else {
37       is error, error recovery or exit;
38     }
39   } else {
40     is error, do error recovery;
41   }
42 }
```

```
9         pop lookahead symbol;
10      update S and L;
11    } else if A is reduce then {
12      output code for this reduction;
13      r1 ← LHS symbol of reduction A;
14      r2 ← RHS symbol count of A;
15      pop r2 states from state_stack,
16        update current state S;
17      A_tmp ← action for (S, r1);
18      push target goto state A_tmp to state_stack;
19    } else if A is reduce/reduce conflict then {
20      let k = 2;
21      while true do
22         L_next ← next lookahead;
23         if L_next == EOF then
24             Report error and exit;
25         else
26             In LR(k) parsing table, find entry A_next ← ((S, L), L_next);
27             if A_next is reduce/reduce conflict then
28                 L ← L_next;
29                 k ← k + 1;
30             else
31                 do reduce;
32                 break out of while loop;
33    } else if A is accept then {
34      if next token is EOF then {
35        is valid accept, exit;
36      } else {
37        is error, error recovery or exit;
38      }
39    } else {
```

IV. Storing the Parsing Table

- Store LR(1) Parsing table
- Store LR(k) Parsing table

Table 3. Storage arrays for the parsing machine in Hyacc parse engine.

Array name	Explanation
yyfs[]	List the default reduction for each state. If a state has no default reduction, its entry is 0. Array size = *n*.
yyrowoffset[]	The offset of parsing table rows in arrays yyptblact[] and yyptbltok[]. Array size = *n*.
yyptblact[]	Destination state of an action (shift/goto/reduce/ accept). If yyptblact[i] is positive, action is 'shift/goto', If yyptblact[i] is negative, action is 'reduce', If yyptblact[i] is 0, action is 'accept'. If yyptblact[i] is -10000000, labels array end. Array size = *p*.
yyptbltok[]	The token for an action. If yyptbltok[i] is positive, token is terminal, If yyptbltok[i] is negative, token is non-terminal. If yyptbltok[i] is -10000001, is place holder for a row. If yyptbltok[i] is -10000000, labels array end. Array size = *p*.
yyr1[]	If the LHS symbol of rule i is a non-terminal, and its index among non-terminals (in the order of appearance in the grammar rules) is *x*, then yyr1[i] = -*x*. If the LHS symbol of rule i is a terminal and its token value is *t*, then yyr1[i] = *t*. Note yyr1[0] is a placeholder and not used. Note this is different from yyr1[] of Yacc or Bison, which only have non-terminals on the LHS of its rules, so the LHS symbol is always a non-terminal, and yyr1[i] = x, where x is defined the same as above. Array size = *r*.
yyr2[]	Same as Yacc yyr2[]. Let x[i] be the number of RHS symbols of rule i, then yyr2[i] = x[i] << 1 + y[i], where y[i] = 1 if production i has associated semantic code, y[i] = 0 otherwise. Note yyr2[0] is a placeholder and not used. This array is used to generate semantic actions. Array size = *r*.
yynts[]	List of non-terminals. This is used only in debug mode. Array size = number of non-terminals + 1.
yytoks[]	List of tokens (terminals). This is used only in debug mode. Array size = number of terminals + 1.
yyreds[]	List of the reductions. Note this does not include the augmented rule. This is used only in debug mode. Array size = *r*.

Table 4. LR(k) parse engine arrays (in addition to LR(1) storage arrays)

Array name	Explanation
yyfs[]	
yyrowoffset[]	
yyptblact[]	
yyptbltok[]	
yyr1[]	
yyr2[]	
yynts[]	
yytoks[]	
yyreds[]	
yy_lrk_k	The maximum value of k for this LR(k) grammar.
yy_lrk_rows[]	The number of rows in each LR(k) parsing table. For each parsing table, there may be multiple states, and each state may have multiple tokens.
yy_lrk_cols[]	Each row starts with two fields for each (state, token) pair, followed by one entry for each token.
yy_lrk_r[]	The actual values in each LR(k) parsing table. The first two entries are for the (state, token) pair. This is followed by (index, value) pairs, where index is the index of a token in the yyPTC[] array, and value is the action for this token: i) -2 means reduce/reduce conflict, ii) a positive number means no conflict and is the corresponding production's ruleID, iii) -1 labels the end of each row.
yyPTC[]	The values of parsing table column tokens.

IV. Storing the Parsing Table

- E.g., Given grammar G1:
  - E → E + T | T
  - T → T * a | a

IV. Storing the Parsing Table

- LR(1) Parsing Table for G1

State	*$*	+	*	*a*	*E*	*T*
0	0	0	0	s3	g1	g2
1	a0	s4	0	0	0	0
2	r2	r2	s5	0	0	0
3	r4	r4	r4	0	0	0
4	0	0	0	s3	0	g6
5	0	0	0	s7	0	0
6	r1	r1	s5	0	0	0
7	r3	r3	r3	0	0	0

IV. Storing the Parsing Table

- Storage tables in Hyacc LR(1) Parsing Engine for grammar G1

```
#define YYCONST const
typedef int yytabelem;

static YYCONST yytabelem yyfs[] = {0, 0,
0, -4, 0, 0, 0, -3};

static YYCONST yytabelem yyptbltok[] = {
97, -1, -2, 0, 43, 0, 43, 42, -10000001, 97,
-2, 97, 0, 43, 42, -10000001, -10000000};

static YYCONST yytabelem yyptblact[] = {
3, 1, 2, 0, 4, -2, -2, 5, -4, 3,
6, 7, -1, -1, 5, -3, -10000000};

static YYCONST yytabelem yyrowoffset[] = {
0, 3, 5, 8, 9, 11, 12, 15, 16};

static YYCONST yytabelem yyr1[] = {
     0,    -1,    -1,    -2,    -2};
static YYCONST yytabelem yyr2[] = {
     0,     6,     2,     6,     2};
```

IV. Storing the Parsing Table

- Storage tables in Hyacc LR(1) Parsing Engine for grammar G1

```
#ifdef YYDEBUG

typedef struct {char *t_name; int t_val;}
yytoktype;

yytoktype yynts[] = {
  "E", -1,
  "T", -2,
  "-unknown-", 1   /* ends search */
};
yytoktype yytoks[] = {
  "a", 97,
  "+", 43,
  "*", 42,
  "-unknown-", -1   /* ends search */
};
char * yyreds[] = {
  "-no such reduction-"
  "E : 'E' '+' 'T'",
  "E : 'T'",
  "T : 'T' '*' 'a'",
  "T : 'a'",
};
#endif /* YYDEBUG */
```

V. Precedence & associativity

- Same as in Yacc and Bison
- Can help to resolve s/r and r/r/ conflicts.
  - Default:
    - In case of shift/reduce conflict, choose shift
    - In case of reduce/reduce conflict, use the one whose production appears first in the grammar
  - Customized: use precedence and associativity rules
- customized precedence and associativity rules
  - Specified using %left, %right, %nonassoc

V. Precedence and associativity

- Define Associativity
  - %left: left associative
    - Reduce is left associative
  - %right: right associative
    - Shift is right associative
  - %nonassoc: no associativity

V. Precedence and associativity

- Define Precedence
  - For tokens
    - 2 tokens declared in the same precedence declaration have the same precedence
    - If defined in different precedence declarations, the one declared later has higher precedence
    - If a token is declared by %token, it has no associativity, and its precedence level is 0
    - If a token is declared by %left or %right, then with each declaration, the precedence is increased by 1
  - For rules
    - A rule gets its precedence level from its last terminal token
    - Context-dependent precedence: defined with %prec T, where terminal token T is declared in %left or %right earlier

V. Precedence and associativity

- Resolve conflicts using precedence and associativity
  - Resolve shift/reduce conflict
    - E.g., a state containing below 2 configurations have a s/r conflict on '+':
      - E → E + E . {+}
      - E → E . + E {+}
    - To choose between shift (on token t) and reduce (by rule r).
      - Precedence of rule r is greater than the precedence of token t: use reduce
      - Token t and rule r have equal precedence, but the associativity of rule I is left: use reduce
      - If either the token or the rule has no precedence: use shift
      - In other conditions and by default: use shift.
  - Resolve reduce/reduce conflict
    - E.g., a state containing below 2 configurations have a r/r conflict on '+':
      - E → E + E . {+}
      - E → E . {+}
    - To choose, use the reduce whose rule appears first in grammar.
  - There can be no shift/shift conflict, which is an error.

VI. Error Handling

- Same as in Yacc
  - Token **error** is reserved for error handling
  - The parser will detect a syntax error when it is in a state where the action associated with the lookahead symbol is **error**
  - A semantic action can cause the parser to initiate error handling by executing the macro YYERROR
  - When the parser detects a syntax error, it calls **yyerror** with the character string syntax error as its argument. The call will not be made if the parser is still recovering from a previous error.
  - The parser is considered to be recovering from a previous error until the parser has shifted over at least three normal input symbols since the last error was detected or a semantic action has executed the macro **yyerrok**.
  - The macro **yyerrok** in a semantic action will cause the parser to act as if it has fully recovered from any previous errors.
  - The macro **yyclearin** will cause the parser to discard the current lookahead token.

VII. Data Structures

- Symbol table
  - Implemented as a hash table (open chaining)
  - Store pre-calculated attribute, for O(1) performance
  - No insertion and find, no deletion needed.
  - Symbol node definition:
    - Symbol: pointer to the string
    - Value: integer representing the symbol
    - Symbol_type: terminal, non-terminal or none
    - TerminalProperty: precedence and associativity
    - Seq: parsing table column number
    - ruleIDList: list of rules whose LHS is this symbol

VII. Data Structures

- Careful choice of linked list/dynamic array/static array
  - Linked list: used when the number of entries is not known, and only sequential access is needed
  - Dynamic array: used when indexed access is needed for fast retrieval
  - Static array: used when the number of entries in known
  - Sometimes these are used together for the same set of objects.
  - E.g., States are stored in all 3, which to use depends on situation:
    - State_collection stores states as a linked list
    - State_array stores all states too, it is a dynamic array of state pointers to states in State_collection linked list
    - To search fast, a hash table is used to store hash values of the states

VII. Data Structures

- Extensive use of hash table
- No size limit for any data structure
  - Can grow until use up all memory.
  - Exception:
    - Artificial upper limit of 512 char for symbol length, and
    - Artificial upper limit of 65536 for the number of UnitProdState objects in the UPE algorithm

VIII. Future work

- Support these Yacc directives:
  - %nonassoc, %union, %type
- Make it re-entrant
- Add parse engines in Java, C++, C# etc

Questions?

Kihei Beach @ Maui - Hawaii